Lección 2: Comprobación de Tipos

1. Introducción
2. Sistemas de tipos
3. Representación de tipos
   - Expresiones de tipos
   - Árboles
4. Comprobación de tipos
   - Equivalencia
   - Conversión
   - Coerción
   - Inferencia
5. Perspectiva

Lecturas:

- Scott, capítulo 7
- Aho, capítulo 6
- Fischer, sección 10.1
- Holub, sección 6.3
- Bennett, capítulo 9
- Cooper, sección 4.2

1. Introducción

```
int *(*s[10])[];
```

- ¿tipo de esta declaración?
- ¿es legal?
- ¿cuánto espacio ocupa **s**?

```
var A, B: array(1..10)
          of boolean;
A := B;
```

- ¿es esto legal en ADA?

```
procedure p(...);
begin
   .... q(...); ....
end;
procedure q(...);
begin
   .... p(...); ....
end;
```

- ¿cómo comprobar esta declaración?

```
int i, v[10];
...
v[i] = 1;
```

```
var i : 1..10;

i := 10;
j = succ(i);
```

- ¿puede detectarse estos problemas durante la compilación?

```
typedef struct {
        int tipo;
        struct nodo *izq;
        struct nodo *der;
    } nodo;
```

- ¿cómo comprobar esta declaración?

Análisis Semántico

Nombres

```
if (r > i) goto L1
```

- `r, i:` declaradas,
- `L1:` aparecer una vez

```
loop A
  ...
end A;
```

- `A:` aparecer al comienzo y final (o no aparecer)

Tipos

```
... r = i(); ...
... if (a == b) ...
```

- `i`: declarado como función
- valor devuelto por `i` compatible con el tipo de `r`
- `a` y `b:` comparables

```
... r = i; ...
... i = r; ...
```

- convertir `i` a real
- convertir `r` a entero

```
int i, j, k;
float r, s, t;

k = i + j;
j = i + r;
t = r + s;
```

- determinar qué tipo de operación llevar a cabo

Lugar en el programa

```
break;
```

```
continue;
```

- sólo dentro de un bucle

```
goto L1;
```

- sólo en el mismo bloque

Comprobación de tipos

1. **Verificar** que los tipos y valores asociados a los objetos de un programa se utilizan de acuerdo con la especificación del lenguaje.
2. **Detectar** conversiones implícitas de tipos para efectuarlas o insertar el código apropiado para efectuarlas

- Almacenar información relativa a los tipos de los objetos

¿representación?

- Aplicar las reglas de verificación de tipos

¿cómo y cuándo aplicarlas?

- **Equivalencia:** determina cuándo dos objetos pueden considerarse del mismo tipo.
- **Compatibilidad:** determina cuándo un objeto de cierto tipo puede ser usado en un cierto contexto.
- **Inferencia:** derivación del tipo de un objeto a partir de sus componentes.
- **Conversión:** permitir y efectuar un cambio de tipo.
- **Coerción:** conversión automática de un tipo a otro.

2. Sistemas de tipos

1. Tipos primitivos

- **booleano:** un byte, 0 (false), 1 (true)

```
int i = 2;
...
if (i) ...
```

- **caracter:** un byte, ASCII
  Java: dos bytes, UNICODE
- **entero:** dos bytes...

```
char  c;
int   i;
short s;
long  l; ...
```

- **real:** coma fija, coma flotante
- **racionales?**
- **complejos?**

2. Constructores

- **enumerados:** orden total

```
type dia is (lun, mar,
             mie, jue,
             vie, sab,
             dom);

type fs is  (sab, dom);
```

- **subrangos:** intervalos
- **registros:** tuplas
- **vectores:** secuencias

```
var s: string[80];
...
write(s);
```

- **conjuntos:** selección
- **punteros:** referencias
- **listas:** sec. sin indexado
- **ficheros:** sec. con pos.
- **funciones:** genera un valor

Constructores: ortogonalidad

- Cada elemento debe ser **independiente** de los demás
- Deben poder usarse en cualquier **combinación**
- Todas las combinaciones deben tener **sentido**

Sentencias (Algol):

```
a := if b < c then d
                  else e;
...
a := begin
       f(b);
       g(c)
     end;
...
g(c);
2 + 3;
```

C:

```
if (a == b) ...
if (a = b) ...
```

Idealmente, los tipos deberían ser ortogonales (hace más clara la programación.)

Tipos :

- ¿subrangos de reales?
- ¿conjuntos de caracteres?
- ¿registros de registros?
- ¿vectores de registros?

- ¿conjuntos de reales?
- ¿ficheros de ficheros?
- ¿vectores con índices reales?
- ¿vectores con índices registros?

Sistemas de tipos

3. Reglas

- de **equivalencia** de tipos:
  cuándo los tipos de dos objetos son iguales
- de **compatibilidad** de tipos:
  cuándo puede usarse un valor en un contexto
- de **inferencia** de tipos:
  determinar el tipo de una expresión con base en el tipo de sus elementos

- La **complejidad** semántica del lenguaje depende de la variedad de tipos primitivos y de constructores.

- La **rigidez** semántica depende de la flexibilidad en la aplicación de las reglas de utilización y conversión de tipos.
  - BASIC, Fortran: simples y rígidos
  - C: complejo, pero poco rígido
  - ADA: complejo y rígido

Según cómo y cuando se aplican estas reglas...

Lenguajes estáticamente tipados

- Los tipos de los objetos se determinan durante la compilación. Esto se hace a través de la declaración de todos los objetos antes de su utilización. **Ejemplos: Java y C.**

Lenguajes dinámicamente tipados

- Los tipos se determinan durante la ejecución. **Ejemplos: Python, Lisp, Snobol4.**

Lenguajes débilmente tipados

- El programador puede obviar, y en algunos casos ignorar, el tipo de los objetos que maneja. **Ejemplo: C**

Lenguajes fuertemente tipados

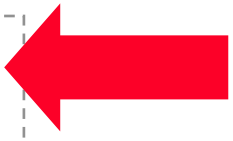
- Las reglas de utilización de tipos se aplican estrictamente. **Ejemplos: Java, ADA.**

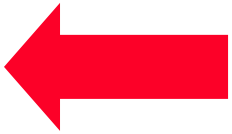
Lenguajes DEBILMENTE tipados: C

- **Tipos primitivos:** `char, int, float, double ...`
- **Constructores:** `*, [], struct, ()`
- **Declaración:** `'specifier' ('declarator')*`

```
sp decl*
int i;          /* entero                          */
int *j;         /* puntero a entero                */
int *k();       /* funcion dev puntero a entero    */
int (*l)();
int *m[9];
int (*n)[];
int *o()[];
int **p();
int *q()();
int **r[][];
int *(*s[10])[];
int *(*f())();
```

```
main(int argc, char *argv[])
{
  char *w[];
  ...
}
```





3: array size missing in `w'

```
int f(int v[][10])
{
  v[1][2] = 8;
}
```

ok

C: lenguaje débilmente tipado

- Las funciones pueden tener un número variable de parámetros:

```
{
  int i, j, k, f();

  j = f (i);
  k = f (i, j);
}
```

- La compatibilidad es muy amplia:

```
enum {
       lun, mar, mie,
       jue, vie, sab,
       dom
     } dia;
...
dia = 4;
dia = 40;
```

- Se permite la conversión:

```
int  *i;
char *j;
...
j = (char *) i;
```

- Se efectúa coerción:

```
float  f(int i)
{
  int j;
  ...
  return j;
}
```

```
main()
{
  int i; char a;

  i = 540; a = i;
  printf ("%d\n", 540);
  printf ("%d\n", i);
  printf ("%d\n", a);
}
```

FUERTEMENTE tipados: ADA

- **Tipos primitivos:** `character, integer, boolean, float, fixed`
- **Constructores:** `array, record, access...`

- Hay conversión:

```
var r : float;
    i, j : integer;
...
r := float (2*J);
j := integer (1.6);
i := integer (-0.4);
```

- No hay coerción:

```
if (r + 1 > 0) then
...
```

- El concepto de equivalencia es muy restrictivo:

```
var A, B: array(1..10)
          of boolean;
...
A := B;
```

- La comprobación es más sencilla (está mas claro lo que hay que hacer).
- La sobrecarga aumenta la flexibilidad:

```
type dia is (lunes,
             martes,
             miercoles,
             jueves,
             viernes,
             sabado,
             domingo);

type festivo is (sabado,
                 domingo);
```

3. Representación de Tipos

1. Para describir la estructura de un tipo en un lenguaje de programación se puede utilizar una notación funcional llamada ***expresión de tipos.***

Una *expresión de tipos* es:

- Un tipo *primitivo*
- Un *constructor* aplicado a una expresión de tipos

C:

```
tipo
array(componente, indice)
record(campo1,...,campon)
pointer(tipo)
function(arg1, ..., argn, tipo)
```

```
int i;              i: int
int *j;             j: pointer(int)
int *k();           k: function(pointer(int))
int (*l)();
int *m[9];
int (*n)[];
int *o()[];
int **p();
int *q()();
int **r[][];        r?
int *(*s[10])[];    s?
```

Árboles de Tipos

2. Una expresión de tipo también puede describirse gráficamente como un ***árbol de tipos***.
   - Las hojas son tipos o valores primitivos.
   - Los nodos interiores son constructores.

- Ejemplo (Pascal):

```
var a: array[1..10] of array[1..20] of integer;
```

- Expresión de tipos:

- Árbol de tipos:

Tipos en la Tabla de Símbolos (Tabla de Atributos)

- Ejemplo (PASCAL P4): registro con campos variantes

```
type
 IdPtr = ^Identifier;

 Identifier = record
   Name: Alpha;
   Llink, Rlink: IdPrt;
   IdType: TypePtr;
   Next: IdPtr;
   case Class : IdClass of
    Constant: ( Value: ValueType );
    TypeName  ();
    Variable: ( Vkind: IdKind;
                Vlevel: LevelRange;
                Vaddr: AddressRange
              );
    Field: ( Offset: AddressRange );
    Proc, Func: (
      case PFDeclKind : DeclKind of
       Standard: (...);
       Declared: (...)
    )
   end;
```

Parte fija:
- Nombre
- Inf. de acceso
- Clase
- Tipo

Tipos en la Tabla de Símbolos (Tabla de Atributos)

- Ejemplo (PASCAL P4): registro con campos variantes

Parte variable:
- Tamaño
- .....

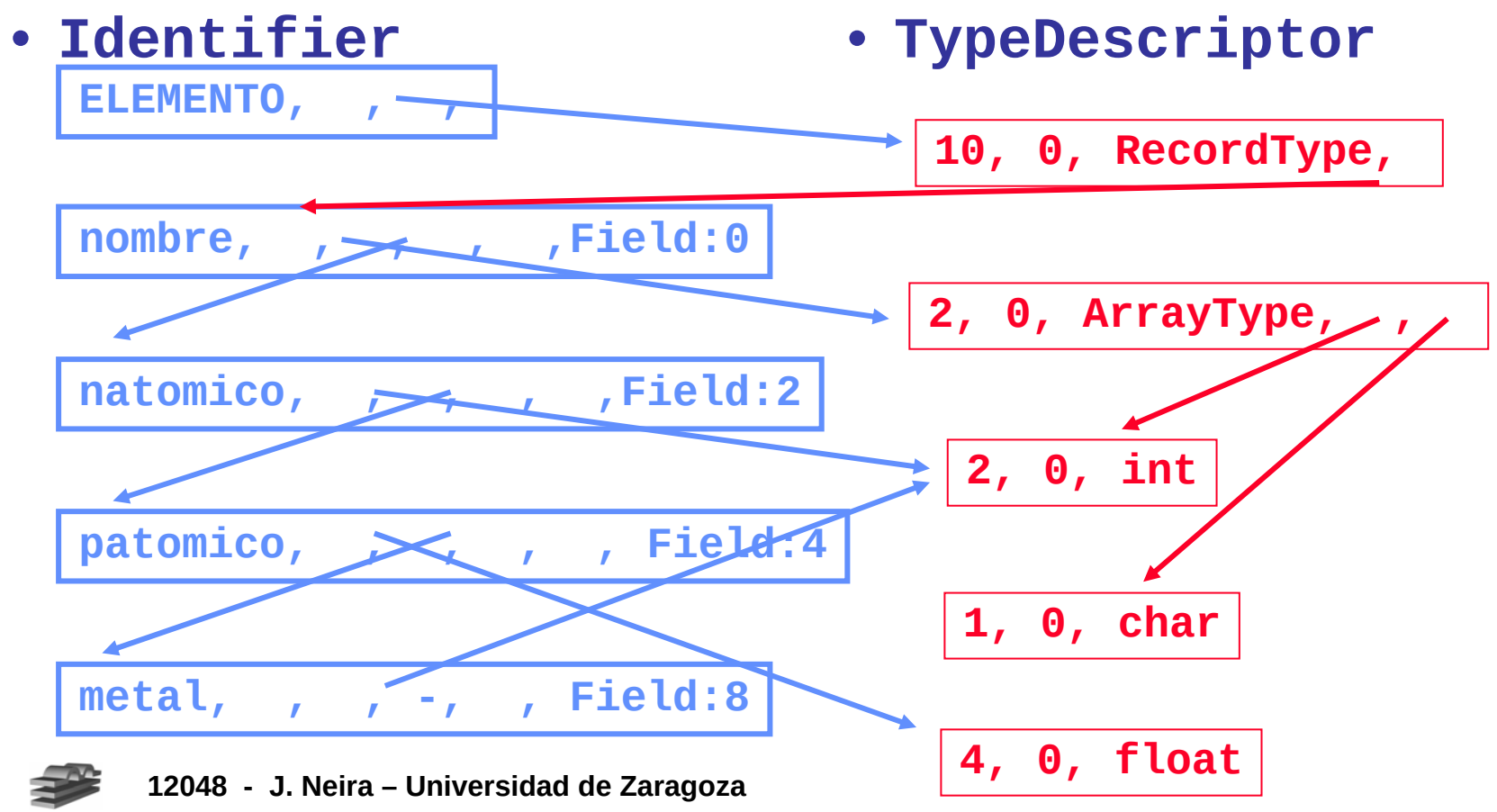
```
TypePtr = ^TypeDescriptor

TypeDescriptor = record
   Size: AddressRange;
   PackedFlag: boolean;
   case Form : TypeForm of
    Scalar: (
      case ScalarKind : DeclKind of
       Declared: ( First: IdPtr );
       Standard: ( )
    );
    Subrange: ( SubBType: TypePtr;
                Min, Max: Value
              );
    Pointer: ( PtrBType: TypePtr );
    SetType: ( SetBType: TypePtr );
    ArrayType: (
      IndexType, ElementType: TypePtr
    );
    RecordType: ( FirstField: IdPtr );
    FileType: ( FileBaseType: TypePtr )
  end;
```

Representación de ELEMENTO

¿representación?

```
typedef struct {
   char  nombre[2];
   int   natomico;
   float patomico;
   int   metal;
} ELEMENTO;
```



4. Comprobación de Tipos

- **Comprobación Estática:** llevada a cabo durante la compilación
  - Toda la información necesaria debe estar disponible
  - Programas más eficientes (no se genera código)

```
#define I 5

short  i;
char   v[10];

i = 10*32767;

...

v[2*I] = 'b';

...

v[4] = 1000;
```

- **Comprobación Dinámica:** llevada a cabo durante la ejecución.
  - Programas menos eficientes (desventaja de los intérpretes).

```
procedure Compute is
 K : Integer :=
     Integer'Last;
begin
 K := K + 1;
end Compute;
```

  - Referencia a variables sin valor asignado.

```
var i : integer;

write(i);
...
```

  - Punteros a NULL.

```
var p : ^integer;

^p := 100;
...
```

Equivalencia

- **Equivalencia estructural:** dos tipos son equivalentes si sus componentes lo son, y están organizadas de igual manera
- La definición exacta depende del lenguaje.

```
type r1 = record
   i : integer;
   f : real;
end;

type r2 = record
   f : real;
   i : integer;
end;
```

?

```
type r1 = record
   i, j : integer;
end;

type r2 = record
   i : integer;
   j : integer;
end;

type r3 = record
   a : integer;
   b : integer;
end;
```

?

```
type v1 = array[1..10]
                of char;
     v2 = array[0..9]
                of char;
```

?

```
type persona = record
   nombre,
   direccion : string[80];
end;

type empresa = record
   nombre,
   direccion : string[80];
end;
```

?

Equivalencia estructural

- **Método usual de comprobación:** algoritmo recursivo de comparación de árboles.
- C: comprobación de tipos estructural:

```
{
  int (*f)();
  int (*g)();

  f = g;
}
```

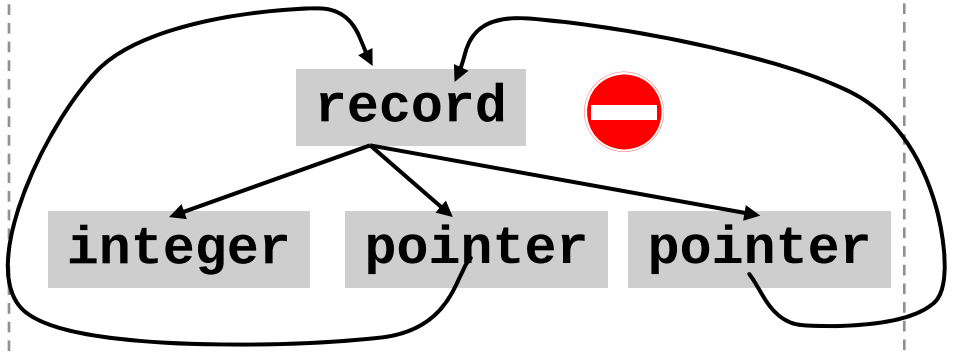
- ¡Excepto para registros!

```
struct {
     int i; char c;
} a;
struct {
     int i; char c;
} b;
...
a = b;
```

- ¿por qué? tipos recursivos:

```
typedef struct {
        int tipo;
        struct nodo *izq;
        struct nodo *der;
    } nodo;
```

- Árbol de tipos:



¡El método de comprobación debe utilizar los nombres!

- Expresión de tipos:

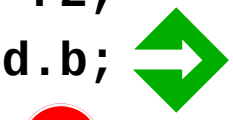
```
nodo:  record(int,
          pointer(nodo),
          pointer(nodo))
```

Equivalencia

- **Equivalencia Nominal:** dos tipos son equivalentes si tienen el mismo nombre

```
type r1 : record
            b : integer;
            c : real;
          end;

     r2 : record
            b : integer;
            c : real;
          end;
var a : r1;
    d : r2;
...
a.b := d.b;
a := d;
```



- El programador diferencia los tipos a través de los nombres:

```
type metros = real;
     millas = real;

var d : metros; l : millas;

d := l;
```

- **Pascal:** a cada definición implícita se le asigna un nombre diferente

```
var a : record
          b : integer;
          c : real;
        end;
    d : record
          b : integer;
          c : real;
        end;
...
a := d;
```

- **ADA:** la compatibilidad es MUY restringida

```
var A, B : array(1..10)
                of BOOLEAN;
....
A := B;
```

Conversión

- El programador modifica **explícitamente** el tipo de una expresión; puede ocurrir que:
  1. Los tipos tengan la misma representación a bajo nivel; entonces es una operación puramente conceptual.

En ADA:

```
type tt = 0..10;
var t : tt;
    i : integer;
...
i := integer(t);
```

En C (*casting*):

```
float *pr;
int *pi;
...
  pr = (float *) pi;
  pi = (int *) pr;
```

  2. Tienen diferente representación a bajo nivel (estas operaciones pueden no ser reversibles)

Conversión en PASCAL:

```
var r : real;
    i : integer;
    c : char;
..
  i := trunc (r);
...

  i := ord (c);
```

Evitar la conversión en C:

```
float r;
char c;
...
c = *((char *) &r);
```

Coerción

- El compilador modifica el tipo **automáticamente**.

PASCAL:

```
var a : real;
    i : integer;
..
    a := i;
```

C:

```
float f()
{
    int i;
    char c;
    ...
    c = i;

    return i;
}
```

ADA:

```
no existe
```

- Tipos genéricos:

```
typedef void *ELEMENTO;
typedef struct nodo {
    ELEMENTO dato;
    struct nodo *sig;
} NODO, *pNODO;
```

- **A favor:**
  facilita la abstracción y extensibilidad de los programas, facilitando la incorporación de tipos nuevos.
- **En contra:**
  riesgos de seguridad. Los lenguajes modernos tienden a alejarse de la coerción.

Inferencia

- Determinar el tipo de un objeto a partir de sus componentes:

```
type Ta = 0..20;
     Tb = 10..20;

var a : Ta,
    b : Tb;

...
   c := a + b;
```

  - No puede ser **Ta**, ni **Tb**.
  - ¿nuevo tipo?

- PASCAL: del tipo base
- ADA: deriva un tipo anónimo

```
+:
r.min := o1.min + o2.min;
r.max := o1.max + o2.max;

-:
r.min := o1.min - o2.max;
r.max := o1.max - o2.min;
```

- ML es especialmente complejo, el compilador siempre debe inferir el tipo:

```
fun circum(r) =
    r * 2.0 * 3.14159;
...
circum(7)

fun cuadrado(x) = x * x;
```

- Provee una forma inmediata de polimorfismo:

```
fun comparar (x, p, q) =
  if x = p then
     if x = q then ”ambos”
     else ” primero”
  else
     if x = q then ”segundo”
     else ”ninguno”;
```

5. Perspectiva

- Comprobación semántica más profunda: **`man lint`**

```
 1 #include <stdio.h>
 2
 3 main ()
 4 {
 5   int i, j, k;
 6   char c;
 7   float *r;
 8
 9   f (5, 4);
10   printf ("%s\n", f (5.0));
11   r = (float *) &c;
12   goto L1;
13   i = f (0);
14   L1: i = j;
15 }
16
17 int f (n)
18 int n;
19 {
20  if (n > 1)
21   return n * f (n-1);
22  else ;
23 }
```

"lint attempts to detect features of the named C program files that are likely to be **bugs**, to be **non-portable**, or to be **wasteful**. It also performs **stricter type checking** than does the C compiler."

```
(9) (10) f: variable # of args
(10) f, arg. 1 used inconsistently
(9) f returns value which is
      sometimes ignored
(11) possible pointer alignment
     problem
(20) function f has return(e);
     and return;
(13) statement not reached
(13) i set but not used
(14) j may be used before set
(5) k unused in function main
```

5. C .vs. ADA

http://libre.act-europe.fr/Software_Matters

```
enum  Alert_Type {LOW, MEDIUM, HIGH, VERY_HIGH};

void handle_alert (enum  Alert_Type  alert) {
  switch (alert) {
    case LOW:
      activate_camera ();
    case MEDIUM:
      send_guard ();
    case HIGH:
      sound_alarm ();
  }
}

void process_alerts () {
  handle_alert (2);
  …
```

- Compila, pero ¿tiene problemas?

5. C .vs. ADA

http://libre.act-europe.fr/Software_Matters

- Olvidas los break
- Olvidas algún caso importante
- 2 y High no son lo mismo

```
void handle_alert (enum
Alert_Type  alert) {
  switch (alert) {
    case LOW:
      activate_camera ();
      break;
    case MEDIUM:
      send_guard ();
      break;
    case HIGH:
      sound_alarm ();
      break;
    case VERY_HIGH:
      alert_police ();
      break;
  }
}
void process_alerts () {
  handle_alert (HIGH);
```

5. C .vs. ADA

http://libre.act-europe.fr/Software_Matters

- ADA es más seguro
- Verifica todos los casos
- No hace falta break
- Solamente puedes usar el tipo Alert_type

```
type  Alert_Type  is  (LOW,
MEDIUM, HIGH, VERY_HIGH);

procedure Process_Alert (Alert :
Alert_Type) is
begin
  case Alert  is
    when LOW     =>
      Activate_Camera;
    when MEDIUM  =>
      Send_Guard;
    when HIGH    =>
      Sound_Alarm;
    when VERY_HIGH   =>
      Alert_Police;
  end case;
end Process_Alert;
```

5. C .vs. ADA

http://libre.act-europe.fr/Software_Matters

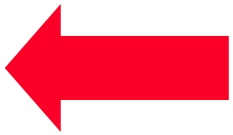
```
typedef   int  Time;
typedef   int  Distance;
typedef   int  Speed;
…
const  Speed  SAFETY_SPEED = 120;
…
void  increase_speed (Speed s);
…
void  check_speed (Time t, Distance d) {
  Speed  s = d/t;
  if  (s < SAFETY_SPEED)
    increase_speed (t);
}
void  perform_safety_checks () {
  Time    t  =  get_time ();
  Distance   d     =  get_distance ();
  …
  check_speed (d, t);
}
```

- Compila, pero ¿tiene problemas?

5. C .vs. ADA

http://libre.act-europe.fr/Software_Matters

```
typedef   int  Time;
typedef   int  Distance;
typedef   int  Speed;
…
const  Speed  SAFETY_SPEED = 120;
…
void  increase_speed (Speed s);
…
void  check_speed (Time t, Distance d) {
  Speed  s = d/t;
  if  (s < SAFETY_SPEED)
    increase_speed (t);
}
void  perform_safety_checks () {
  Time    t   =  get_time ();
  Distance   d      =  get_distance ();
  …
  check_speed (d, t);
}
```



- Compila, pero ¿tiene problemas?

5. C .vs. ADA

http://libre.act-europe.fr/Software_Matters

```
SAFETY_SPEED : constant Integer := 120;
…
procedure Increase_Speed (S : Integer);
…
procedure  Check_Speed (T : Integer; D : Integer) is
  S : Integer := D / T;
begin
  if  S < SAFETY_SPEED then
    Increase_Speed (T);
  end if;
end Check_Speed;

procedure  Perform_Safety_Checks is
  T   : Integer  :=  Get_Time;
  D   : Integer  :=  Get_Distance;
begin
  …
  Check_Speed (D, T);
end Perform_Safety_Checks;
```

- NO compila